HW2

1. (20%) Use Gaussian elimination with partial pivoting to solve the following equations (given as the augmented matrix). Are any row interchanges needed?

$$\left[\begin{array}{ccc|c} 3 & 1 & -4 & 7 \\ -2 & 3 & 1 & -5 \\ 2 & 0 & 5 & 10 \end{array}\right]$$

ANS

$$\mathbf{x} = \begin{bmatrix} 3.206 \\ 0.238 \\ 0.714 \end{bmatrix}$$

```
❯ python p1.py
Solution: [3.206 0.238 0.714]
Number of row interchange: 0
```

Code

```
import numpy as np

def Gaussian_Elimination(A, b):
    row_interchange = 0
    n = len(A)
    x = np.zeros(n)

    for i in range(n):
        # Partial pivoting
        pivot = i + np.argmax(np.abs(A[i:, i]))
        if (pivot != i): # Need row interchange
            row_interchange += 1
            A[[i, pivot]] = A[[pivot, i]]
            b[i], b[pivot] = b[pivot], b[i]

        # Gaussian Elimination
        for j in range(i + 1, n): # iterate each row between
(i, m)
            mul = -A[j][i] / A[i][i]
            A[j] = [a + mul * b for a, b in zip(A[j], A[i])]
```

```
            b[j]+= mul * b[i]

    # back substitution
    for i in range(n - 1, -1, -1):
        x[i] = (b[i] - np.dot(A[i][i+1:], x[i+1:])) / A[i][i]
        x[i] = round(x[i], 3)

    print(f"Solution: {x}")
    print(f"Number of row interchange: {row_interchange}")

A = np.array([[3, 1, -4], [-2, 3, 1], [2, 0, 5]])
b = np.array([7, -5, 10])


Gaussian_Elimination(A, b)
```

2. (20%) A system of two equations can be solved by graphing the two lines and finding where they intersect. (Graphing three equations could be done, but locating the intersection of the three planes is difficult.) Graph this system; you should find the intersection at (6, 2).

$$0.1x + 51.7y = 104,$$
$$5.1x - 7.3y = 16,$$

   a. Now, solve using three significant digits of precision and no row interchanges. Compare the answer to the correct value.

   b. Repeat part (a) but do partial pivoting.

   c. Repeat part (a) but use scaled partial pivoting. Which of part (a) or (b) does this match, if any?

ANS

```
❯ python p2.py
Part A solution: [10.    1.99]
Part B solution: [6.02 2.01]
Part C solution: [6.02 2.01]
```

   a. the difference between the approximate solution and true solution is [4. -0.01]

   b. it's very close to true solution with difference [0.02, 0.01]

   c. the result is the same with part b

3. (20%) Given system A:

$$A = \begin{bmatrix} 2 & -1 & 3 & 2 \\ 2 & 2 & 0 & 4 \\ 1 & 1 & -2 & 2 \\ 1 & 3 & 4 & -1 \end{bmatrix}$$

Find the LU equivalent of matrix A that has 2's in each diagonal position of L rather than 1's.

Ans

```
❯ python p3.py
L:
[[ 2.          0.          0.          0.        ]
 [ 2.          2.          0.          0.        ]
 [ 1.          1.          2.          0.        ]
 [ 1.          2.33333333 -6.          2.        ]]
U:
[[ 1.         -0.5         1.5         1.        ]
 [ 0.          1.5        -1.5         1.        ]
 [ 0.          0.         -1.          0.        ]
 [ 0.          0.          0.        -2.16666667]]
```

$$L = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 2 & 2 & 0 & 0 \\ 1 & 1 & 2 & 0 \\ 1 & 2.333 & -6 & 2 \end{bmatrix}, U = \begin{bmatrix} 1 & -0.5 & 1.5 & 1 \\ 0 & 1.5 & -1.5 & 1 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -2.167 \end{bmatrix}$$

Code

```
import numpy as np

def LU_factorization(A):
    n = len(A)
    L = np.eye(n) * 2
    U = np.zeros((n, n))

    for j in range(n):
        U[0][j] = A[0][j] / 2
        for i in range(1, j + 1):
            sum_u = sum(L[i][k] * U[k][j] for k in range(i))
            U[i][j] = (A[i][j] - sum_u) / 2
        for i in range(j, n):
            sum_l = sum(L[i][k] * U[k][j] for k in range(j))
            L[i][j] = (A[i][j] - sum_l) / U[j][j]
```

```
    return L, U

A = np.array([[2, -1, 3, 2], [2, 2, 0, 4], [1, 1, -2, 2], [1,
3, 4, -1]])

L, U = LU_factorization(A)

print("L:")
print(L)
print("U:")
print(U)
```

4. (20%) Solve this system with the Jacobi method. First rearrange to make it diagonally dominant if possible. Use [0, 0, 0] as the starting vector. How many iterations to get the solution accurate to five significant digits?

$$\begin{bmatrix} 7 & -3 & 4 & 6 \\ -3 & 2 & 6 & 2 \\ 2 & 5 & 3 & -5 \end{bmatrix}$$

ANS

```
❯ python p4.py
Solution: [-0.14332486 -1.37459432  0.71987058]
Iteration times: 34
```

Code

```
import numpy as np

def Jocobi_Method(A, b, TOL):
    n = len(A)
    # Check for diagonally dominant
    for i in range(n):
        d = np.argmax(A[i, 0:])
        A[[i, d]] = A[[d, i]]
        b[i], b[d] = b[d], b[i]

    x = np.zeros(n)
    diff = 1
    cnt = 0
    while (diff > TOL):
```

```
        cnt += 1
        x_prev = np.copy(x)
        for i in range(n):
            x[i] = (b[i] - np.dot(A[i][0:], x_prev[0:]) + A[i]
[i] * x_prev[i]) / A[i, i]
        diff = np.sum(np.abs(x - x_prev))
    print(f"Solution: {x}\nIteration times: {cnt}")

A = np.array([[7, -3, 4], [-3, 2, 6], [2, 5, 3]])
b = np.array([6, 2, -5])

Jocobi_Method(A, b, 1e-5)
```

5. (20%) Repeat Problem 4 with the Gauss-Seidel method. Are fewer iterations required?

ANS

```
❯ python p5.py
Solution: [-0.14332299 -1.37459376  0.71986976]
Iteration times: 14
```

Code

```
import numpy as np

def GaussSeidel_Method(A, b, TOL):
    n = len(A)
    # Check for diagonally dominant
    for i in range(n):
        d = np.argmax(A[i, 0:])
        A[[i, d]] = A[[d, i]]
        b[i], b[d] = b[d], b[i]

    x = np.zeros(n)
    diff = 1
    cnt = 0
    while (diff > TOL):
        cnt += 1
        x_prev = np.copy(x)
        for i in range(n):
            x[i] = (b[i] - np.dot(A[i][0:], x[0:]) + A[i][i] *
x[i]) / A[i, i]
```

```
        diff = np.sum(np.abs(x - x_prev))
    print(f"Solution: {x}\nIteration times: {cnt}")

A = np.array([[7, -3, 4], [-3, 2, 6], [2, 5, 3]])
b = np.array([6, 2, -5])

GaussSeidel_Method(A, b, 10, 5)
```

6. (25%) This 2×2 matrix is obviously singular and is almost diagonally dominant. If the right-hand-side vector is $[0, 0]$, the equations are satisfied by any pair where $x = y$.

$$\begin{bmatrix} 2 & -2 \\ -2 & 2 \end{bmatrix}$$

   a. What happens if you use the Jacobi method with these starting vectors: [1, 1], [1, -1], [-1, 1], [2, 5], [5, 2]?

      ANS

```
Part A, Jocobi Method:

Starting vector: [1, 1]
solution: [1.0, 1.0]

Starting vector: [1, -1]
solution diverge

Starting vector: [-1, 1]
solution diverge

Starting vector: [2, 5]
solution diverge

Starting vector: [5, 2]
solution diverge
```

      Except for [1, -1], other starting vectors can not find a solution.

   b. What happens if the Gauss-Seidel method is used with the same starting vectors as in part (a)?

      ANS

```
Part B:, Gauss-Seidel Method

Starting vector: [1.0, 1.0]
solution: [1.0, 1.0]

Starting vector: [1.0, -1.0]
solution: [-1.0, -1.0]

Starting vector: [-1.0, 1.0]
solution: [1.0, 1.0]

Starting vector: [2.0, 5.0]
solution: [5.0, 5.0]

Starting vector: [5.0, 2.0]
solution: [2.0, 2.0]
```

All starting vectors find a solution, which is different from Jacobi method.

c. If the elements whose values are -2 in the matrix are changed slightly, to -1.99, the matrix is no longer singular but is almost singular. Repeat parts (a) and (b) with these new matrix.

ANS

```
Part C, Jocobi Method:

Starting vector: [1, 1]
solution: [0.0009904930420933113, 0.0009904930420933113]

Starting vector: [1, -1]
solution: [2.4923546448972597e-06, -2.4923546448972597e-06]

Starting vector: [-1, 1]
solution: [-2.4923546448972597e-06, 2.4923546448972597e-06]

Starting vector: [2, 5]
solution: [8.303274282580806e-06, 3.321309713032743e-06]

Starting vector: [5, 2]
solution: [3.321309713032743e-06, 8.303274282580806e-06]
```

```
Part C:, Gauss-Seidel Method

Starting vector: [1, 1]
solution: [0.0004934687472038007, 0.0004910014034677818]

Starting vector: [1, -1]
solution: [-0.0004934687472038007, -0.0004910014034677818]

Starting vector: [-1, 1]
solution: [0.0004934687472038007, 0.0004910014034677818]

Starting vector: [2, 5]
solution: [0.0004961528561156321, 0.0004936720918350539]

Starting vector: [5, 2]
solution: [0.0004941681959703543, 0.0004916973549905025]
```

Jacobi method get some solution that $x \neq y$ and the number is very small. However Gauss-Seidel method still get every solution that x is very close to y.

Code

```
import numpy as np

def Jocobi_Method(A, b, x, TOL):
    n = len(A)
```

```
    # Check for diagonally dominant
    for i in range(n):
        d = np.argmax(A[i, 0:])
        A[[i, d]] = A[[d, i]]
        b[i], b[d] = b[d], b[i]

    diff = 1
    cnt = 0
    while (diff > TOL and cnt < 1e5):
        cnt += 1
        x_prev = np.copy(x)
        for i in range(n):
            x[i] = (b[i] - np.dot(A[i][0:], x_prev[0:]) + A
[i][i] * x_prev[i]) / A[i, i]
        diff = np.sum(np.abs(x - x_prev))
    if (cnt >= 1e5):
        print("solution diverge")
    else:
        print(f"solution: {x}")

def GaussSeidel_Method(A, b, x, TOL):
    n = len(A)
    # Check for diagonally dominant
    for i in range(n):
        d = np.argmax(A[i, 0:])
        A[[i, d]] = A[[d, i]]
        b[i], b[d] = b[d], b[i]

    diff = 1
    cnt = 0
    while (diff > TOL and cnt < 1e5):
        cnt += 1
        x_prev = np.copy(x)
        for i in range(n):
            x[i] = (b[i] - np.dot(A[i][0:], x[0:]) + A[i]
[i] * x[i]) / A[i, i]
        diff = np.sum(np.abs(x - x_prev))
    print(f"solution: {x}")


starting_vectors =  [[1, 1], [1, -1], [-1, 1], [2, 5], [5,
```

```
2]]

# Part A
print("\nPart A, Jocobi Method:")
for x in starting_vectors:
    print(f"\nStarting vector: {x}")
    A = np.array([[2, -2], [-2, 2]])
    b = np.array([0, 0])
    Jocobi_Method(A, b, x, 1e-5)

print("\nPart B:, Gauss-Seidel Method")
for x in starting_vectors:
    print(f"\nStarting vector: {x}")
    A = np.array([[2, -2], [-2, 2]])
    b = np.array([0, 0])
    GaussSeidel_Method(A, b, x, 1e-5)

# Part C
print("\nPart C, Jocobi Method:")
for x in starting_vectors:
    print(f"\nStarting vector: {x}")
    A = np.array([[2, -1.99], [-1.99, 2]])
    b = np.array([0, 0])
    Jocobi_Method(A, b, x, 1e-5)

print("\nPart C:, Gauss-Seidel Method")
for x in starting_vectors:
    print(f"\nStarting vector: {x}")
    A = np.array([[2, -1.99], [-1.99, 2]])
    b = np.array([0, 0])
    GaussSeidel_Method(A, b, x, 1e-5)
```